REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE 27 Dec 95	3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
A Visualization Tool for Engineering Vector Analysis	

6. AUTHOR(S)

Byron L. Miranda

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) AFIT Students Attending: Washington University	8. PERFORMING ORGANIZATION REPORT NUMBER 95-146

| 9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) DEPARTMENT OF THE AIR FORCE AFIT/CI 2950 P STREET, BLDG 125 WRIGHT-PATTERSON AFB OH 45433-7765	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for Public Release IAW AFR 190-1 Distribution Unlimited BRIAN D. Gauthier, MSgt, USAF Chief Administration	12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

DTIC SELECTED JAN 0 5 1996 F

Original contains color plates: All DTIC reproductions will be in black and white

19960104 132

14. SUBJECT TERMS			15. NUMBER OF PAGES 52
			16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

DTIC QUALITY INSPECTED 1

DISCLAIMER NOTICE
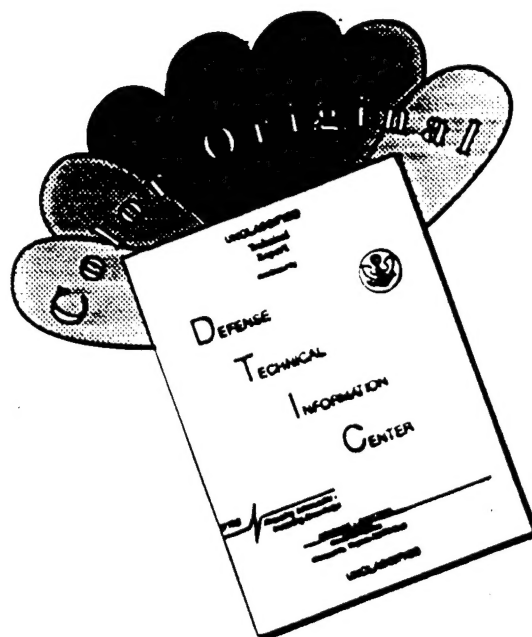
THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF COLOR PAGES WHICH DO NOT REPRODUCE LEGIBLY ON BLACK AND WHITE MICROFICHE.

A Visualization Tool for Engineering Vector Analysis

by

Byron L. Miranda

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science in Engineering

University of Washington

1995

Approved by _____
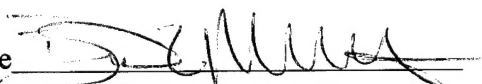
Chairperson of Supervisory Committee

Program Authorized
to Offer Degree_____ Civil Engineering _____

Date___ 12/11/95 _____

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this thesis is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Any other reproduction for any purposes or by any means shall not be allowed without my written permission.

Signature _____

Date___11____DEC__95_____

TABLE OF CONTENTS

LIST OF FIGURES

Number

ACKNOWLEDGMENTS

Chapter 1

INTRODUCTION

This thesis presents a prototype tool designed to support an interactive, visual approach to the learning of vector analysis. The objective was to construct a prototype environment in which vectors, scalars and points could be entered, manipulated and observed in three-dimensional form in order to investigate new mechanisms for linking mathematical abstractions to intuitive, geometric thinking at an early stage in a student's education.

Visualizing three-dimensional problems is difficult for many students when the concept is first presented, but this ability is fundamental to many fields of engineering, and therefore must be mastered [McCuistion 1991]. The motivation for this work is to determine a means of aiding students in entry-level engineering courses develop their ability to visualize and analyze in 3-D, specifically in regards to vector analysis.

Vector analysis is typically an integral part of an engineering student's introduction to 3-D thinking. When first introduced, vectors and the results of vector operations often are presented geometrically so that the student may gain some sense of what they actually "look" like. This is often hindered, however, by the two-dimensional limitations of paper and chalkboard, and by the time required to complete the mathematical calculations and graphically render them in a meaningful way. Computers offer the attraction of allowing rapid calculations; more importantly, they allow quasi-real-time graphical presentation of the results of these calculations.

Therefore, students could be able to see the results of their work as it is done in a richer and dynamic fashion.

Educational software commonly used for teaching mathematical concepts, such as Matlab and Maple, have well-developed mathematical and graphical capabilities; however, these programs do not explicitly provide the environment that was desired in this project. Both Matlab and Maple require a fair amount of knowledge before use, and are not aimed at the entry-level engineering student. Individuals newly introduced to vectors presumably will posses little familiarity with the matrices fundamental to the use of both these programs. More significantly, these mathematical languages do not inherently provide the type of interface desired in this project: real-time, integrated visual input and feedback.

The background information intrinsic to this project is presented first. This includes discussing and demonstrating both the need for visualization skills, and the difficulty in developing them. Current methods of presenting 3-D graphics are described, followed by the issues faced in creating effective user interfaces. The issues governing the design of an effective 3-D visualization program are presented, followed by the approaches taken in developing prototype software to explore these design issues. Some example problems demonstrating the functionality of the prototype software precede the concluding remarks.

Chapter 2

BACKGROUND

2.1 Visualization

When first introduced to three-dimensional concepts, it can be difficult for students to visualize the topic under discussion. An example is the parallelogram law for the addition of forces; it states that two forces acting on a particle can be replaced by the diagonal resultant of a parallelogram drawn with the sides equal to the two given forces. In two-dimensions this is a simple concept for students to grasp and practice. However, the simplicity rapidly diminishes when the law is extended to a third
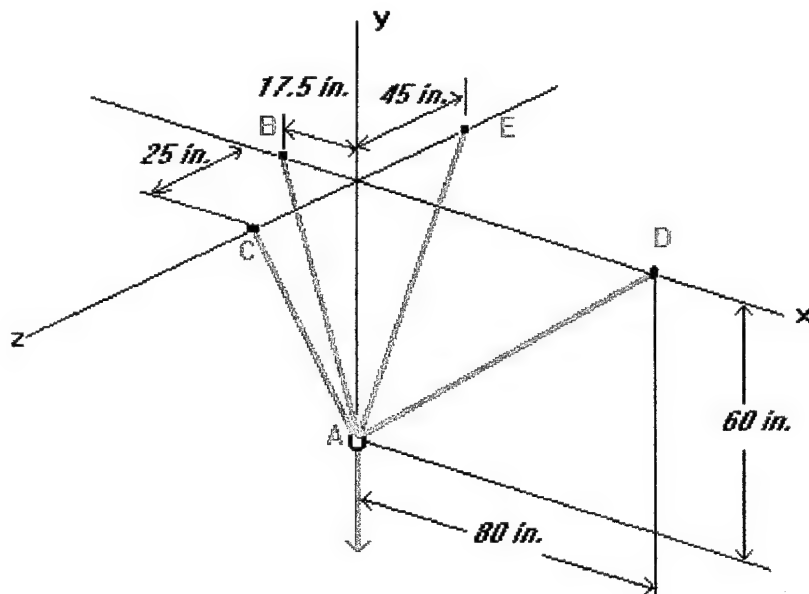
Figure 2.1 Problem involving vector analysis [Beer and Johnston 1992]

dimension. The validity of the law is no less true, but the difficulty of drawing the sides meaningfully with the additional dimension can overwhelm the student.

Further difficulties are encountered when the student begins working problems. A typical problem requiring vector analysis is shown in Figure 2.1. Although the static image is clearly labeled and contains the information required to reach a solution, a student who does not already have well developed visualization skills may have difficulty understanding the drawing. Some drawings may be so difficult to interpret visually that the vector information must be presented numerically. But if the student does not possess a clear understanding of vector analysis, the additional information does not aid in problem visualization. This results in the student achieving an analytical solution without ever gaining a feel for the underlying physical action.

Any two-dimensional format available to an instructor is limited in its ability to present three-dimensional information. This problem is exacerbated with beginning students who may not have well-developed visualization abilities, and thus may be unable to fully comprehend presented information. Physical Cartesian coordinate systems, made of plastic or wood, offer the advantage of a tangible three-dimensional environment in which to display similarly designed vectors. However, while helpful for the student, such tools have obvious limitations. There is the simple logistical problem of getting enough models for each student to use, and the physical problems associated with accuracy, spatial manipulations and certain geometric representations (i.e. how to represent a point). Further difficulties are encountered in determining meaningful methods of showing vector operations.

Computers, with their ability to both rapidly conduct mathematical operations and present sophisticated graphics could provide an excellent means of helping students visualize three-dimensional vector analysis. The greatest potential benefit of computers in teaching vector analysis is the ability to display information in what Wiebe [1992] refers to as a dynamic/presentational combination. The user is offered a high degree of

flexibility in problem solving and visualization because of the ability to interactively view graphic information in quasi-real-time.

2.2 Three-Dimensional Graphics

Humans use a system of perceptual mechanisms to understand and perceive spatial relations and the size and position of complex objects in three-dimensional environments, rather than any single mechanism [Haber 1990]. The purpose of a three-dimensional computer graphic system is to provide the perceptual cues required for the user to easily interpret the geometry and position of some rendered object(s). The reality of designing such a system is to provide *just* enough of these cues to assure user comprehension.

A large number of techniques exist for creating meaningful, computer-generated 3-D graphics; only the most common will be discussed in this thesis. After this general overview, the cues and techniques used in the prototype developed for this project will be presented. The more common techniques for rendering the required visual cues include three-dimensional transformations; hidden line and surface removal with Z-buffers; the Phong and Gouraud shading methods; ray tracing; adding shadows; and surface texturing.

2.2.1 Three-Dimensional Transformations

Using a computer for graphical rendering presents the immediate problem of representing a 3-D object on a 2-D computer monitor. A means of displaying the third dimension of depth must be found. This is generally done in Cartesian coordinates (x, y, z), using a 3 x 3 matrix for rotation and scaling, and adding a fourth dimension for translation and perspective. Therefore, a point is represented with a four-dimensional position vector of (x, y, z, 1), and is multiplied by a 4 x 4 generalized transformation matrix to achieve a given view. This operation allows the user to scale, rotate, or translate an object about the screen as desired.

2.2.2   Z-buffers

When observing a scene, objects that overlap others are interpreted as lying in the foreground between the observer and more distant objects. This powerful perceptual mechanism allows the determination of relative position between multiple objects. Similarly, the amount of an observed object that is visible is obviously dependent upon the observer's position. The computer must determine which surfaces and lines can be seen from this position, and those that must be hidden. While it may seem clear in computer graphics that not all of an object can be seen, "hidden-line" and "hidden-surface" algorithms to accomplish this rapidly become computationally expensive [Haber 1990].

The least complex solution to this problem is the Z-buffer algorithm, applied after completion of the viewing transformation. An imaginary three-dimensional screen space is created, where the (x, y) values are the pixel coordinates, and the z value is the viewing space depth. At each point (x, y), a search is conducted for the point with the minimal z value. The search is implemented with a Z-buffer which holds for the given (x, y) point the smallest z value so far encountered. Thus it is determined which object is closer to the viewer and should be drawn, and which should be hidden. The Z-buffer is advantageous in that it is a simple algorithm to implement, but unfortunately it requires a tremendous amount of memory [Watt 1989]. Modern hardware used in graphics rendering systems typically solves this problem by providing special Z-buffer memory. However, designers have developed more complex algorithms that are less expensive in terms of memory requirements; this allows the Z-buffering to occur in software, without expensive hardware modifications [Gallagher 1995].

2.2.3   Phong Shading

Depending upon the complexity of a rendered object, shading can be the primary cue allowing the user to understand its three-dimensional shape. For example,

with no shading, a sphere and circle of identical diameter are indistinguishable [Haber 1990]. Reflection models are thus used to represent light interactions with a surface, in terms of surface properties and lighting. In computer graphics, these models allow the 2-D screen renderings of 3-D objects to mimic reality to an acceptable level [Watt 1989]. The purpose the application will serve, and the hardware available, determine what is an acceptable level. The Phong method is one of the most commonly used to achieve realistic shading.

The Phong reflection method simulates reflected light with three terms: ambient, diffuse, and specular. Ambient light illuminates surfaces not directly lit by the light source; it is a result of reflections from all the surfaces in a scene. In the Phong method, the ambient term is constant, used both to keep objects from appearing too dark and to approximate indirect illumination. The diffuse term refers to the fact that most observed objects do not emit light, but absorb and reflect a portion of it [Watt 1989]. Phong assumes perfect diffusion, where light is scattered equally in all directions, and the amount of reflected light is independent of the observer's position, dependent only upon the amount of light received. The specular term simulates the behavior of light on glossy materials such as polished metal. All light from a source reflects off a glossy surface in one direction, known as the mirror direction, and can be seen only when viewed from this direction. The Phong technique maximizes the specular term at the mirror direction, falling off rapidly as a function of the angle away from that direction [Fournier and Buchanan 1995].

The objects in the view are modeled as a system of polygons, where the geometric information is known at the vertices. In an effort to speed up what can become quite expensive shading computations, shade is calculated only at the polygon vertices, and interpolated across the interior [Fournier and Buchanan 1995]. In the Phong method, the vertex normals are calculated by averaging the normal vectors of the surfaces sharing the vertex. The normals are then interpolated across the interior, and the light intensity is calculated for each pixel in the interior from these normals.

This is done to avoid the problem of producing specular highlights which are totally dependent upon the positions of the modeled polygons. The Phong method gives good results by avoiding many of the problems inherent in other algorithms, but has the disadvantage of requiring intensive calculations, due to the necessity of calculating the vertex normals and the intensity of each pixel.

2.2.4 Gouraud Shading

Gouraud shading is similar to Phong in that it calculates shading by interpolating the polygon vertex values across the surface. But instead of calculating vertex normals, the Gouraud scheme calculates the light intensity at each vertex, and interpolates these intensities over the surfaces. The prime advantage of this method is that it is far less computationally expensive than the Phong method, and is generally effective, particularly with diffuse surfaces. However, specular highlights are dependent upon relative polygon position; this can cause problems, particularly in animated sequences when the polygons shift position, due to unnatural shifting of the highlights [Watt 1989].

2.2.5 Ray Tracing

Salmon and Slater [1987] describe ray tracing, a technique which allows more complex and sophisticated 3-D graphics. Most graphics use ray casting; for each screen pixel, a ray is traced from the eye position through the pixel into the scene. Intersections of the ray with all objects are calculated, the object with the smallest Z value is the one shown. Ray tracing is much more complex; after the first object intersection, the ray is reflected and traced recursively until it either passes out of the scene or returns to the original light source. If the surface is transparent, an additional ray is created and refracted through the surface, and traced to completion. This method allows the introduction of transparency, reflection, and shadow to a rendering,

in a reasonably simple manner. Ray tracing is, however, extremely expensive in computational terms, particularly because of the need to calculate the intersections.

2.2.6 Shadowing

Shadows present cues on geometry and relative position. However, adding shadows to an image typically has been regarded as improving the quality of already adequate computer graphics, primarily because calculations can become extremely expensive computationally. Although there are a great many shadow algorithms in existence, none has emerged as being clearly superior to the others [Watt 1989]. One of the simplest approaches is the shadow Z-buffer technique, which is easily integrated into Z-buffer based rendering schemes. This method requires a separate shadow Z-buffer for each light source. As the scene is drawn, each pixel is checked against the Z-buffer values to determine if the point is in shadow or not. As the number of additional light sources increases, massive amounts of memory must be allotted to store the Z-Buffer information.

2.2.7 Texture Mapping

Texture mapping is used to create more complex and interesting three-dimensional images, in addition to removing the somewhat plastic appearance commonly imparted by many shading techniques [Watt 1989]. In simple terms, texture mapping is a process of transforming a texture onto the surface of a 3-D object. There are a great many techniques for detailing the texture. They can be generated from scanned real images, simple algorithms such as checkerboard patterns, or a variety of extremely complex methods [Fournier and Buchanan 1995]. The method selected depends upon the image being represented, the desired detail, and the system hardware.

2.2.8 Approach in This Project

The techniques of providing requisite visual cues, presented above, can generally be categorized in three levels of ascending complexity: 3-D transformations, hidden-line and surface removal, and shading. In the prototype developed for this project, only the first category of visual cues was judged necessary to achieve user comprehension of presented graphics. Visual cues are added to allow viewers full comprehension; a static image may require a significant number of cues to reach this goal. However, dynamic images, particularly those allowing user manipulation, aid in perception by allowing the viewer to observe a scene from differing positions, perspectives, and angles. Certain techniques useful with static images, such as shadowing, can be distracting in a dynamic environment, add greatly to the number of required calculations, and offer little, if any, benefit to the goal of user comprehension. The bottom line in this project was that the finished program should provide the user with rapid visual feedback, using only modest hardware.

The translation/rotation matrix was essentially as described above, though it was done in more explicit terms. The objective was to transform the given three-dimensional points into two-dimensional screen coordinates for plotting. This can be expressed as follows:

$$P_x = (\underline{v}_i \cdot \underline{s}) / (d / f) + s_0 \qquad (2.1)$$

$$P_y = (\underline{v}_i \cdot \underline{t}) / (d / f) + t_0 \qquad (2.2)$$

$$d = n_0 - (\underline{v}_i \cdot \underline{n}) \qquad (2.3)$$

$$\underline{v}_i = -(P_i - O) \qquad (2.4)$$

where:

P_x and P_y are the (x, y) screen coordinates of the point;

P_i is the three dimensional point;

$\underline{s}, \underline{t}, \underline{n}$ are the screen x, y, z directions, are the column vectors taken from the 3 x 3 matrix displayed below;

s_0, t_0, n_0, are the x, y, z screen coordinates of the origin, and

allow for the translation, which was the 4 row and

column in the 4 x 4 matrix described above;

d is the distance the user is from axis, for perspective;

$\underline{v_i}$ is the vector from the origin to the point;

f is the focal length, which gives perspective to drawing.

The rotation matrix, shown below, limited the viewer to rotations in two dimensions.

$$\begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ -\sin\phi\sin\theta & \cos\phi & -\sin\phi\cos\theta \\ \cos\phi\sin\theta & \sin\phi & \cos\phi\cos\theta \end{bmatrix} \qquad (2.5)$$

where:

θ is the angle of rotation about the y-axis; and

ϕ is the angle of rotation about the x-axis.

The allowed rotations can be described as the equivalent of placing the user on a sphere surrounding the viewed object and allowing upward, downward, and sideways motion, but no twisting. This is consistent with how humans observe their surroundings; adding a third rotation did not appear as though it would significantly increase student comprehension, if at all.

The vectors and points were modeled simply with lines and circles and without the implementation of any sophisticated lighting techniques. Cylinders and spheres with complex lighting adding specular reflections could have been used, but only at significant computational cost. The key to creating meaningful computer graphics in this project with low-level three-dimensional techniques was to support animation.

2.3 Animation

Animation is used to convey changing behavior in a display. In designing appropriate animation, the first step is to ensure that the proper information is

presented meaningfully. The developer must then derive an animation technique that is both appropriate for the task and compatible with the hardware used.

2.3.1 Animation Design Principles

In recent work, Stasko [1993] identifies four design principles for animation representing an operation or underlying system application: appropriateness, smoothness, duration/control, and moderation. The user has his or her own mental image of the represented operation; therefore, the animation sequence should appropriately represent this image. For instance, opening or closing a program file is commonly represented by showing a folder opening or closing. These types of animation sequences are used to create what are essentially movies, a series of sequential, non-interactive frames. This project's requirements are somewhat different in that the animation is entirely interactive, but the basic precepts still hold true.

In effective animation, actions and motions must be presented in a manner clear to the viewer, abrupt changes in scene or perspective can be difficult to comprehend. It is therefore desirable to have a continuous, smooth scenario which preserves the context, making it easier to understand. This can be difficult to design because it might not accurately represent the program's function. For example, variables change value instantaneously, but the animated response should provide a gradual transition so as to maintain context and provide a sense of before and after states.

The purpose of an animation sequence determines its duration and mode of control. Non-complex sequences should be brief to avoid user impatience, and there should be enough user control of duration to allow for differences in viewer experience and perception. With interactive animation, duration is not an issue as the sequence occurs only at the user's request. Control should be achieved by the least complex and most intuitive method available that still permits all required interactions.

The final design principle a developer must follow is that of moderation. Once the capabilities and attractiveness of animation are grasped, the temptation may grow

to add more and more of it to an application. However, overly sophisticated animation can cause user confusion if too much information is presented. Successful animation communicates and informs; it is not intended merely to impress the user.

2.3.2 Animation Implementation

There are a wide variety of implementation schemes available to the designer, ranging from extremely simple line drawing methods to highly complex algorithms. Method selection is based primarily upon how sophisticated the animation requirements are, and what the hardware will support. One of the simpler methods of representing motion is to erase the entire screen, then redraw the image at its new location. However, drawbacks to this method appear if multiple objects are to be displayed. Because the objects are being erased and redisplayed to the same window, and because they must be redrawn in some order, the cycling becomes noticeable to the user in the form of a distracting screen flicker.

Screen flicker can be eliminated, without changing the algorithm, through the use of double-buffering. At any one time, only one buffer is displayed on the monitor, the other is hidden and available for drawing operations. Therefore, the display can be erased and the objects sequentially redrawn in their new positions while the display is hidden from the user. When the drawing is complete the buffers are rapidly switched; all the objects appear to update simultaneously, without the flicker. Some computers and display devices provide two memory buffers for graphics rendering, although some software provides the additional buffer to avoid special hardware requirements.

2.3.3 Animation in this Project

Motion was represented with the simple re-painting method of erasing and re-drawing the objects in their new positions, without the use of double-buffers. This was done to allow for modest hardware. As previously stated, sophisticated computer animation is often used to create movies. However, the animation required in this

project is intended to provide interactivity between the user and the results of vector operations; this interactive animation interweaves rendering with computation.

2.4 Interface

In a project of this scope, the human-computer interface is the most important consideration. If a program covers a broad scope and is to be used frequently, a user can be assumed willing to learn the capabilities and operations in stages, building knowledge over time. However, with a program covering a narrowly defined scope, as is the case here, it should quickly become apparent how to use the program to its full potential; a user is not going to invest much time learning it. Designers should consider that users are most productive and comfortable in familiar, predicable environments, therefore a Windows product should follow Windows conventions, Macintosh and X-Windows the same.

Apple Corporation published a brief checklist [1989] of considerations a designer should face during the design phase. Many of these considerations are pertinent to projects of this nature, and will be presented below. In addition, design of interactive projects should focus as much as possible on knowledge the user may presumably already possess, by taking advantage of affordances and constraints, as described below.

2.4.1 Basic Principles

The Apple [1989] checklist presents several considerations which should be foremost in the mind of an interface designer. Direct manipulation should be allowed so that users feel control over computer actions. As much as possible, users should select actions from lists of alternatives as opposed to remembering specific commands. Applications should be consistent both within themselves and the rules established in the operating environment. This allows users to accomplish similar actions in different programs, aiding in comfort and ease of learning. Interfaces should be written so the

user initiates all actions, and maintains control of them. Users should be forgiven the inevitable mistakes they will make. That is, a mistake should not result in a program crash, but a clear, concise message explaining what went wrong. Finally, simplicity should be a key concern; the screen should not be crowded, and dialog boxes should not have scores of choices.

2.4.2 Affordances

Eberts [1994] presents the concept of affordances, the actual and perceived properties of an object which determine how the object could be used. For example, the affordance of a knob is for turning, that of a button for pushing. This can be used in product design to improve usability; on a car stereo, the affordance of a knob (turning) provides a logical and obvious means of volume control. What is not so apparent, however, is if the knob is also to be pushed, for an on/off function. The designer may consider it efficient to add the pushing function to the knob, thus saving the addition of an extra button, but this requires the user to read, learn and remember an instruction.

In computer interface design, it is advantageous to minimize the number of instructions the user must learn to operate the software. Real buttons are raised above their surroundings, providing the affordance that they can be moved downward with a push of the finger. The same affordance is often designed in a computer interface by depicting screen buttons raised above their surroundings, which can be moved downward with a push on the mouse button.

Eberts also discusses how sequential affordances can be used in interface design. His natural world example of a sequential affordance is a handle which must be turned to open a door. The handle provides the affordance of grasping, random movement of the hand will provide a tactile affordance that it must be turned downward to open the door. A scroll box demonstrates a similar sequential

affordance; the box on the scrollbar indicates something that can be grabbed with the mouse. Random mouse movement informs the user that the scrollbar can be moved.

2.4.3 Constraints

Another real world concept often used in interface design is that of constraints. In the same way that an electrical plug is designed with one prong larger than the other so that polarity is always correct, an interface should be designed so that the user cannot make mistakes. An obvious example occurs with menu displays; the user is constrained to only those choices displayed.

2.4.4 This Project Interface

The approach taken in this project was to design a user interface that was easy to understand and manipulate, with minimal introduction. The intention was that the focus should be on learning vector analysis, not program operation. It was assumed that the user would be familiar with the environment and conventions established in Windows applications. To take advantage of this familiarity, a mouse and menu-driven format was followed. The affordance of such a design was that the user should already understand how to operate the objects and controls (buttons, scrollbars, menus, etc.) used in the interface.

2.5 Real-Time Graphics

The benefit of a program aiding in the development of visualization skills is dependent upon immediate graphical feedback. The quality and speed of the visual feedback is related to the time required for numerical calculations. Because the focus of this project is to aid in understanding visual representations of vector analysis rather than the underlying algebraic calculations, a real-time graphical display window was required. A vector visualization program should allow the user to conduct a variety of

vector operations with immediately visible results. The use of both low-level 3-D modeling and simple animation techniques minimized computational time spent on calculation and rendering. This permitted quasi-real-time graphical display of user-invoked vector operations.

2.6 Visual Basic

Microsoft Visual Basic was selected for this project because it allows rapid prototyping; it is very easy to build and manipulate graphical user interfaces. Its ease of use allowed interfaces to be completely re-arranged and tested relatively quickly. The amount of programming time expended upon creating graphical controls, etc., was minimized, allowing that time to be spent on higher level considerations. While Visual Basic does not offer near the performance advantages of a language such as C++, this project did not require sophisticated numerical analysis.

It is important to note that while Visual Basic does provide an attractive, easy-to-use environment, this very feature can act against the developer. Visual Basic is designed to do a great many things, and to do them easily; however, if a task is desired that is not explicitly supported by the environment, there is often no alternative to abandoning that task. This simplicity is also limiting in that it does not support double buffering and other sophisticated animation techniques.

Chapter 3

DESIGN ISSUES

3.1 Overview

General issues that must be faced in software design were presented in the previous chapter; this chapter will discuss the specific issues governing design of the prototype. As the prototype was intended primarily as a learning tool for entry-level engineering students, simplicity of design and ease of use were major concerns. The screen set-up should be simple and uncluttered, with a small number of controls present. However, to avoid tediousness, the most commonly used operations should be easily available without having to work through menus. Therefore, with each function/option, there is debate between easy availability and keeping a clear screen. The ultimate goal would be a program which a first-time user could look at, intuitively understand, and immediately begin using. While it was realized that this is not a completely realistic goal for a prototype, it is true that working in that direction should result in a quite accessible finished product; a program where even a first-time user would require only a small amount of time to learn its use.

More specific design guidelines include allowing the user to immediately see the results of a vector operation in a multi-dimensional framework. The points and vectors should be displayed graphically in a meaningful manner allowing user interaction. To take advantage of animation, viewers should be able to observe the results from different angles and distances, thus the angles and relative size of the results should be easy to manipulate.

Additionally, software supporting the teaching of vector analysis requires that three basic data types be available to the user: scalars, vectors and points. This introduces further levels of complexity to the designer. For example, while both points and vectors have three-dimensional descriptions, some means of distinguishing the two is clearly required. Scalars, with no clear means of graphical representation available, offer greater challenges in regards to integration into programs oriented around graphical output.

This project focuses on the fact that in many areas of engineering mechanics there exist multiple means of representing certain abstractions--for example the use of Mohr's circle and index notation to describe and understand the tensorial aspects of stress [Haber 1990]. Similarly, vector analysis can be described with both mathematical and geometric representations. The focal software design issue was therefore to provide an environment in which the user could conduct analytical vector operations with both these representations. This is discussed in greater detail in the following section.

3.2 Multiple Representations

Because the program is intended to aid in visualizing vector analysis, both graphical and numerical representations are required. Existing software often falls towards one of the extreme ends of the simplistic spectrum presented in Figure 3.1, where prompt-only software refers to programs that offer the user only a prompt line for entering data or commands. While programs of this variety can be capable of tremendously complex mathematics, they do not offer equivalent graphical capabilities.
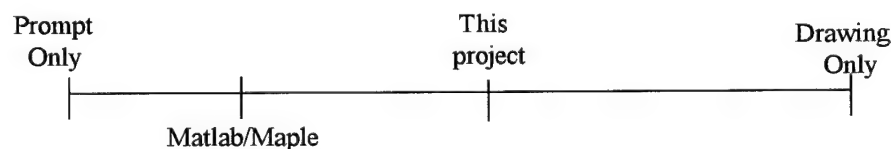
Figure 3.1: Spectrum ranging from prompt-only to graphics-only software.

At the other end of the spectrum are sophisticated drawing programs capable of complex rendering techniques such as ray tracing and texture mapping, but that do not offer the required level of numerical and symbolic analysis.

This project falls towards the middle of the spectrum, requiring only elementary numerical capabilities and a moderate level of graphic sophistication. The challenge lies in determining a successful method of offering both features in a highly interactive environment. Matlab and Maple offer extremely sophisticated numerical and symbolic analysis, as well as high-level graphics. However, these programs fall short as far as this project is concerned because their graphics are not readily interactive. Therefore, achieving the goal of this project, interactive visual feedback of vector analysis, requires a fresh approach different from that taken in existing software.

Chapter 4

THE PROTOTYPE

4.1 Overview

In responding to the design issues posed in the previous chapter, the mouse and menu-driven program shown in Figure 4.1 was developed. As it was assumed that students at this level had at least a basic familiarity with the Windows environment, the
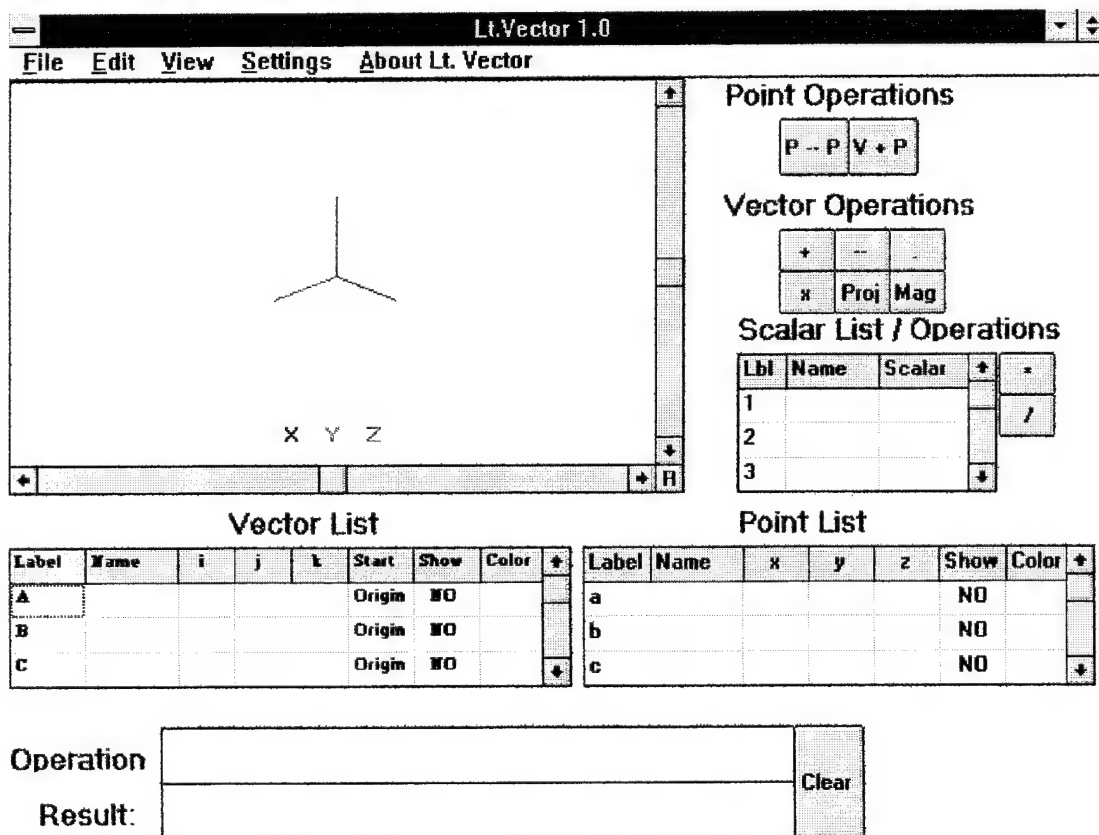
Figure 4.1: Basic layout of Lt. Vector

created program, Lt. Vector, follows the basic conventions and appearances of a Windows application.

Lt. Vector consists of a graphical output window, data management tables, a result window, and several buttons for vector operations. In the following sections, the purpose and use of these components will be discussed in detail.

4.2 Data Input

In a purely graphics-based environment, the user would be able to make entries directly on the screen, using only the mouse. However, there is no simple means of accomplishing this when working in three dimensions, and the problem of scalars having no real graphical meaning would arise as well. Additionally, in a purely graphical environment it would be difficult to permit vectors and points to be hidden at certain times and present at others. For these reasons the hybrid interface in Figure 4.1 was developed.

Type differentiation was accommodated by the creation of spreadsheet-like data grids, a separate one for each type. The user enters data in these grids with mouse and keyboard. While each grid will be discussed in greater detail below, there are some similarities that can be presented here. The user is allowed to enter a name and value for a given data type. Names can be any length (though not all of it will necessarily show), but are not required. Some parsing had to take place to ensure numerical inputs were logical (e.g. entries such as "-1.23a.4-2" had to be prevented). To prevent the program crashing at every user error, incorrect entries are prevented and a message box provides a brief explanation.

4.2.1 Entering Vectors

A primary consideration with computer modeling of vectors is in determining how much information must be entered before the vector will appear. At a minimum, the three components of direction and a base point must be established before a

meaningful rendering can be accomplished. Additionally, it is conceivable that users may not want all vectors to appear on the screen, and may desire the ones that do appear be of differing colors.

In Lt. Vector, once the user has input the three components of a vector, the "Show" column automatically switches to "Yes," the color becomes black, and the vector is immediately drawn. The user can at any time change the "Show" status, but color is not directly a user-specified value, as will be discussed in greater detail below.

Vector base points are an obvious consideration; a vector begins at some pre-defined point in space, such as the origin, or the tip of another vector. For convenience, in Lt. Vector the user is not required to enter a base point prior to rendering, all vectors begin at the origin by default. Once a vector has been entered, the user may opt for a different vector start point by clicking in the "Start" column. This will cause the form shown in Figure 4.2 to appear.

A vector can begin at the origin, tip of a previously input vector, or at a defined point. The point can either be taken from the point list or input directly, in which case it automatically will be added to the list of points. Parsing again occurs to ensure logical user input.

Figure 4.2 Entering vector base points.

4.2.2 Entering Points

Although there is no base point consideration, points are similar to vectors in terms of required input information. For this reason the point list closely resembles the vector list. As with vectors, once the three coordinates have been entered, the point is immediately drawn in black and the "Show" status defaults to "yes."

4.2.3 Entering Scalars

Because there is no graphical representation to provide user feedback, scalars pose a more complex design issue than points and vectors. Whereas points and vectors have multiple representations to aid user understanding, information about scalars must be gleaned entirely from the list. There are also some situations when the ability to change the value of a scalar in a convenient manner would be a desirable feature; for example, a student may wish to observe how a change in magnitude effects the relationship between a set of vectors. In Lt. Vector, the only two columns in the scalar list are name and value. For simplicity of design, the scalar values are input directly from the keyboard.

4.3 View

Aiding in the development of student visualization skills requires that the user possess ready ability to change viewing perspectives. Perspectives over which the user should have control include viewing angle, real and perceived distance (as discussed below), and screen position. The user should also be able to control all objects appearing on the viewer. As was previously discussed, all points and vectors can be hidden by making an appropriate selection on their list; changes in display status of other screen objects are made with the View menu shown in Figure 4.3.

Screen objects can be translated with the scrollbars adjacent to the viewer, and rotations can be accomplished by clicking and dragging the mouse. To avoid user confusion however, it was necessary to place some limits on the axis rotations. To change the rotation limitations, a preset view is selected from the View menu, as displayed in Figure 4.3.

View		
Re-Draw		
Default Angle		
X-Axis		Positive
Y-Axis		Negative
Z-Axis	▶	
Axis InVisible	Ctrl+V	
Axis Labels Off		
Zoom In	Ctrl+I	
Zoom Out	Ctrl+O	
Move In	Ctrl+N	
Move Out	Ctrl+U	
Restore Perspective		
Perspectives Explained		
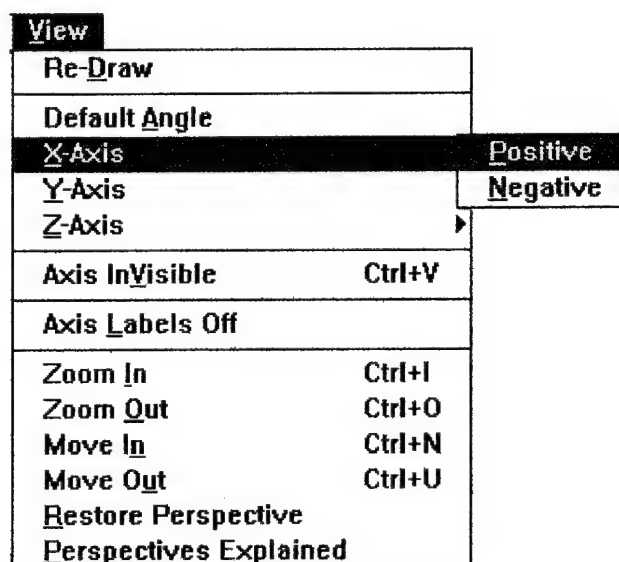
Figure 4.3 Menubar of viewing options.

Further changes in perspective are permitted with zooming and moving. Zooming changes the focal length, and is the equivalent of a stationary photographer zooming with a camera lens. Moving changes the distance between the user and the screen origin; using the same analogy as before, this would be the equivalent of the photographer actually moving in relation to the viewed object. Moving and zooming have similar effects, but if one is changed too much without making adjustments to the other, distortions can occur.

4.3.1 Resetting View

Permitting so many visual changes can result in the user losing perspective and getting lost in the image. While rotations were limited, other restrictions were avoided so as to give greater flexibility. To avoid the problems that can arise with this increased flexibility, reset features to return user to a familiar setting are required. Resetting can occur by storing snapshots of the changing view, and having an undo command available to the user to step back through the stored images. However, as there are multiple perspectives that can be changed independently of one another (e.g. rotation and translation), an effective undo option would be difficult to design.

Resetting in this project was accomplished by returning to the original perspectives. This can be accomplished either by snapping back to the original view, or with an animated sequence showing the incremental changes. In the prototype, the view snaps back when a reset is selected. This method was selected primarily because of the performance limitations of Visual Basic.

To preserve different perspectives, three separate reset features were included. The Default Angle option on the View menu shown in Figure 4.3 restores the original angle to the view, while the Restore Perspective option on the same menu resets the focal length and viewer distance from the origin. Additionally, a small reset button to the lower right of the graphical display, joining the two scrollbars, restores the origin to the center of the viewer.

4.4 Selecting

In order for vector analysis to occur, the specific vectors, points and scalars to be operated with or upon must somehow be distinguished from the others. There are a variety of techniques to set the selections apart visually from other objects on the viewer, but the necessity of showing selection status in the lists was limiting. The technique had to be one which clearly flagged the selection both graphically and on the list. A simple method is to re-draw the selected objects with a double-thick line of a

different color. As color is a property on the lists, this method is easily reflected in both domains.

In Lt. Vector, points and vectors can be selected from the screen by clicking on or near them with the mouse, or directly from the lists by clicking the label column. If more than one selection is desired, the standard convention for multiple selections of pressing the shift key while clicking with the mouse is followed. When the mouse is used on the screen, the nearest displayed object to the click location that has not already been chosen is selected. If an object is selected, it immediately changes color to red. This occurs both on the display where the object is now drawn in red, and in the point and vector lists where the color is changed to red in the appropriate column. The user may select any number of points and vectors at one time.

With no graphic representation available, selecting scalars is more difficult than points and vectors. The two other data types change color when selected; there is no correlating visual cue with scalars. Only one scalar may be selected at a time; to select one, the user clicks on the row of the desired scalar. There is little visual feedback to the user delineating a selected scalar. As will be discussed in greater detail in a following section, however, feedback does appear after each operation.

4.5 Operations

Conducting vector analysis clearly requires that the user be able to conduct several different mathematical operations. To avoid forcing the user to learn the syntax a parser would require, all operations are represented with buttons permanently present on the screen. Operations were grouped by type because each operation dealt primarily with one data type. Before initiating an operation, the user must select the appropriate number of objects. Even without a parser, it is in this area of a program that the greatest potential for user-induced, program-crashing errors exists. It is conceivable that a user might select an incorrect number of vectors for a given operation, for example, or accidentally select points for a vector operation. Therefore, testing had to
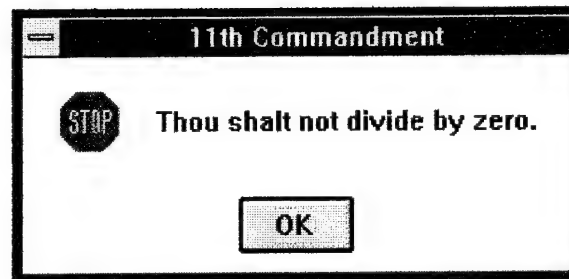
Figure 4.4 Typical error message box; this one prevents scalar division by zero

be conducted to determine the most likely errors a user might make, prevent them from crashing the program, and display an appropriate error message, such as the one in Figure 4.4.
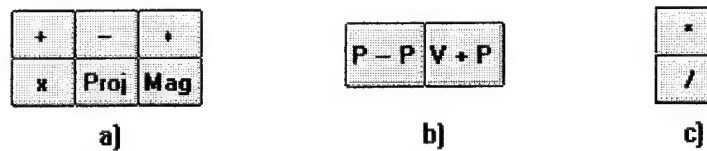
Figure 4.5 a) Vector operations; b) Point operations; c) Scalar operations.

4.5.1 Vector Operations

Lt. Vector permits the six vector operations shown in Figure 4.5a. If no vectors were selected prior to an operation, a message to that effect is shown. If only one vector is selected, the operation is conducted on itself. With add or subtract any number of vectors can be selected, the vectors will be operated upon in the order they were selected. However, with scalar product, cross product, and projection, no more than two vectors can be selected. Magnitude can be calculated only for one vector at a time.

4.5.2 Point Operations

In some problems it may be necessary for a student to determine the position vector between two points in space. Similarly, it can be useful to displace a point by a set distance. The two point operations in Lt. Vector, shown in Figure 4.5b, permit these calculations. Point minus point subtracts one selected point from another to create a vector, while the other operation adds a vector to a point to create a new point.

4.5.3 Scalar Operations

In vector analysis, it is often necessary to multiply or divide a vector by a scalar, as when calculating unit vectors. Therefore, the two scalar operations in Figure 4.5c were added. Both require the selection of a scalar and one vector.

4.6 Results

After an operation has been completed, the results should be presented to the user in a meaningful manner. The most basic form is to display to numerical results of a calculation. A graphical representation either can be shown automatically, or at the

Force Cross Distance
-15.2 i + 18 j + 19.2 k

Figure 4.6 Result window, with the upper portion listing the operation.

request of the user. In order to link symbolic and graphical representations, Lt. Vector automatically displays the results of every operation both graphically and symbolically. The result window, shown in Figure 4.6, is broken into two sections, operation and result. The operation portion displays which calculations were conducted with what selections, so the user can verify that the desired operation was the one that occurred. The result section lists the result of that operation.

Results are then entered into the appropriate data table, with the information from the operation portion going into the "Name" column of the new entry. The scene in the graphical output window is updated to reflect any new points or vectors, which are drawn white, to distinguish them from previously existing objects.

4.7 Program Structure

The overall code structure is closely aligned to the interface, as can be referenced in the code summary presented in Appendix A. The structure can be broken down into general groups associated with the following functionality: graphical output, user-invoked calculations, data management, and monitoring input.

The graphical output required the largest amount of code. This included taking appropriate information from the data tables, transforming the listed 3-D information into 2-D screen coordinates, and allowing the user to change the different perspectives.

User-invoked calculations required a significant quantity of code as well. While a large part of this was caused by allowing the user to select objects from the graphical display window, each vector operation required the creation of a separate subroutine.

The existing data structure stored all data upon the spreadsheet-like grids. This method required code for input, storage, access, and transmission to appropriate modules. As will be discussed in greater detail later in this thesis, there are many areas where the current methods of data management could be changed. These changes could significantly reduce the total volume of code.

Throughout the program, monitoring user input required a significant quantity of code. While there are three separate subroutines checking entries, many other subroutines have internal monitoring routines as well. These internal interpreters catch user-input errors and display appropriate error messages.

Chapter 5

EXAMPLES

5.1 Overview

Several example problems demonstrating the usefulness and limitations of the prototype are presented in this chapter. Following the example problems is a brief critique containing both a discussion of some of the design difficulties encountered with this type of software, and an evaluation of the performance of Lt. Vector with these examples.

5.2 Example 1: Addition of a Series of Vectors

Lt. Vector easily allows the addition or subtraction of series of vectors, as

X Y Z
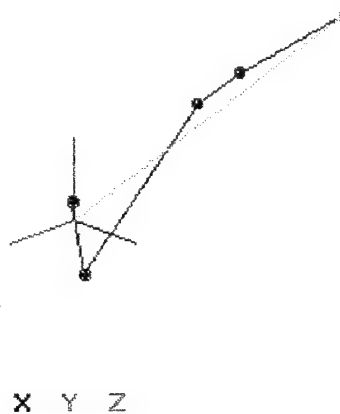
Figure 5.1 Addition of five vectors.

shown in the accompanying figures. Five vectors are entered in the vector list, the first beginning at the origin, the next four beginning at the end of the preceding vector.

The resultant vector begins at the same location as the first vector in the series, and ends at the tip of the last. The user is immediately able to note the relation between the added vectors and the resultant. As shown in Figure 5.2, the view can be manipulated, and extraneous objects removed, to clarify the image as desired to aid in conceptualization.

Figure 5.2: Rotated image showing addition of five vectors.

5.3 Example 2: Geometric Proof with Vectors

Vectors provide a convenient means of investigating geometric theorems, as demonstrated with an example from Thomas and Finney [1989] shown in Figure 5.3. This theorem states that the line segment joining the midpoints of two sides of a triangle is one half the length of, and parallel to, the third side.

This theorem can be demonstrated in Lt. Vector by inputting the triangle endpoints in the point grid. Selecting two endpoints at a time and subtracting them calculates the sides of the triangle as vectors. Entering a "2" in the scalar grid, selecting one side of the triangle at a time and conducting scalar division determines the midpoint vectors of two of the sides. Adding the two midpoint vectors creates the line

segment joining the two midpoints. Dividing the third side by 2 shows the user that the line segment joining the midpoints is parallel to, and half as long as, the third side.
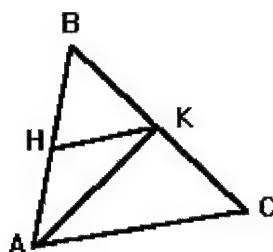
$$HK = AK - AH$$
$$= (AB + 1/2 BC) - 1/2 AB$$
$$= 1/2(AB + BC)$$
$$= 1/2(AC)$$

Figure 5.3: Symbolic proof.

Line F of the vector list shown in Figure 5.4 gives the direction of the vector connecting the midpoints of two sides of the triangle. Line G in the same grid shows the length of the third side divided in half. As expected, they are identical.

Ultimately the goal is for the student to understand the symbolic proof, as shown in Figure 5.4, but using Lt. Vector allows the student to observe the proof demonstrated geometrically and numerically. Furthermore, use of the computer shows that the proof is valid in 3-D, which is not completely apparent with the plane figure in Figure 5.4. The proof could be completed numerically by hand, but the computer is less tedious, and permits the student to observe the results. Lt. Vector does not replace the value of the symbolic proof, but it does provide numerical validation and geometric interpretation. By working through several similar examples with Lt. Vector, the student can gain a more complete understanding of the proof.
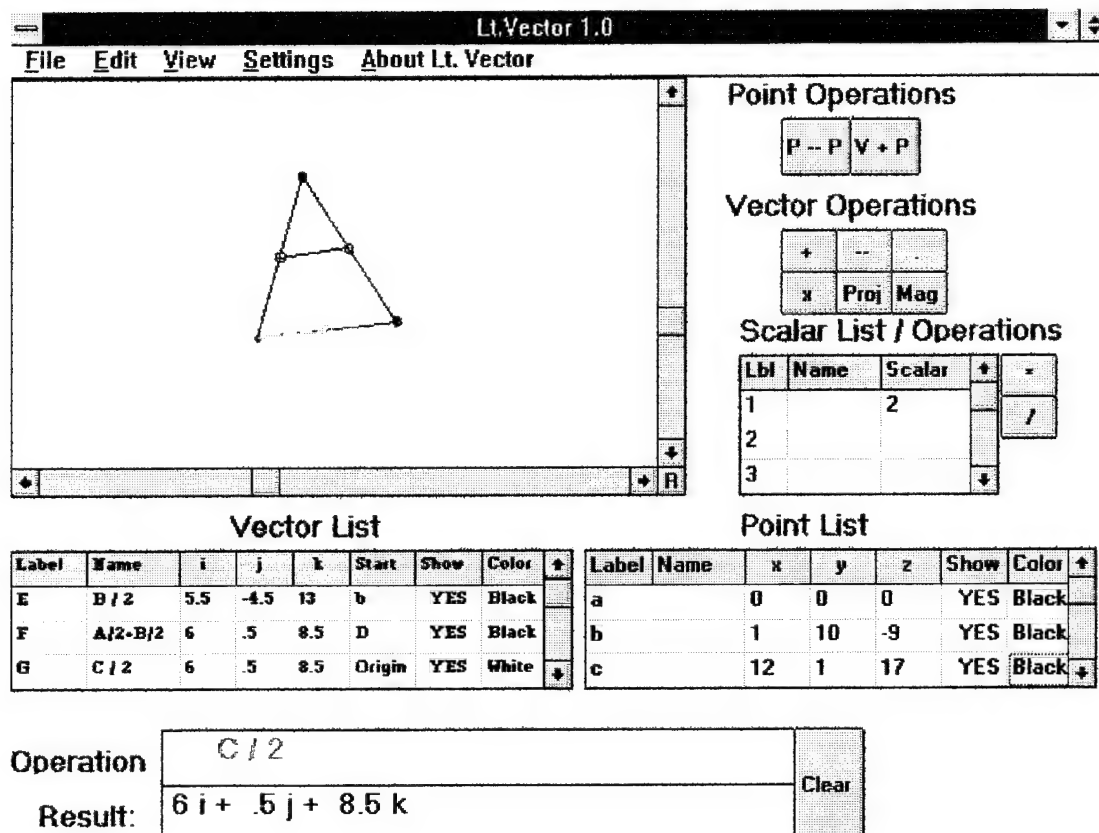
Figure 5.4: Solution to Example 2, using Lt. Vector.

5.4 Example 3: Engineering Mechanics Problem

Lt. Vector is a vector visualization tool, and is not directly capable of solving engineering mechanics problems. Solving systems of equations and operations involving unknowns are not supported. However, Lt. Vector is still useful to the student with these types of problems. Dynamic viewing allows a richer geometric understanding of the problem, and the supplied functions permit rapid vector calculations.

The problem shown previously in Figure 2.1 and reproduced below in Figure 5.5 is a typical introductory engineering mechanics problem requiring vector analysis. The problem states that:

Cable BAC passes through a frictionless ring A and is attached to fixed supports at B and C, while cables AD and AE are both tied to the ring and are attached, respectively, to supports at D and E. Knowing that a 200-lb. vertical load P is applied to ring A, determine the tension in each of the three cables [Beer and Johnston 1992].
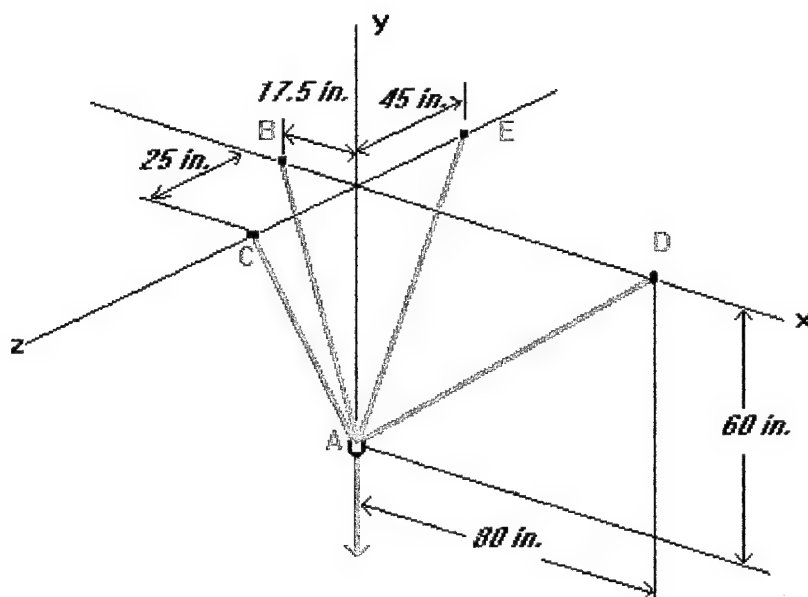
Figure 5.5: Example 5.3 [Beer and Johnston 1992].

Using Lt. Vector with the ring modeled as the origin, the three-dimensional coordinates for the four supports are entered in the point list. The vectors for each support cable are calculated with the point minus point function. The user is now able to manipulate the image as desired to gain a fuller understanding of the geometric situation, as shown in Figure 5.6.

After using the magnitude and scalar division functions to calculate the unit vectors, the student can now reach the exact solution on paper. Enforcing static equilibrium by equating the summed forces to zero allows the student to determine the cable tensions with simple algebraic computations. Alternately, the unit vectors for the

Lt.Vector 1.0

File   Edit   View   Settings   About Lt. Vector

Point Operations

| P -- P | V + P |

Vector Operations

| + | -- | - |
| x | Proj | Mag |

Scalar List / Operations

Lbl	Name	Scalar
2	Mag AC	65
3	Mag AE	75
4	Mag AD	100

Vector List

Label	Name	i	j	k	Start	Show	Color
H	AD / 100	.8	.6	0	Origin	YES	Black
I	AE / 75	0	.8	-.6	Origin	YES	White
J					Origin	NO	

Point List

Label	Name	x	y	z	Show	Color
a	Pt. A	0	0	0	YES	Red
b	Pt. B	-17.5	60	0	YES	Black
c	Pt. C	0	60	25	YES	Black

Operation   AE / 75

Result: 0 i + .8 j + -.6 k   Clear

Figure 5.6: Working through Example 5.3 with Lt. Vector.

support cables can be placed tip to tail with the load vector. By multiplying the unit vectors with scalars, the user can attempt to close the loop, as shown in Figure 5.7.

Figure 5.7: Attempting to close the force vector loop by manipulating scalar multiples.

Attempting to close the force loop by changing the magnitudes of the support vectors will permit the user to converge on the solution. While it would be difficult to reach the exact answers with this technique using the prototype program, it is possible to come close, giving the student a fuller understanding of directio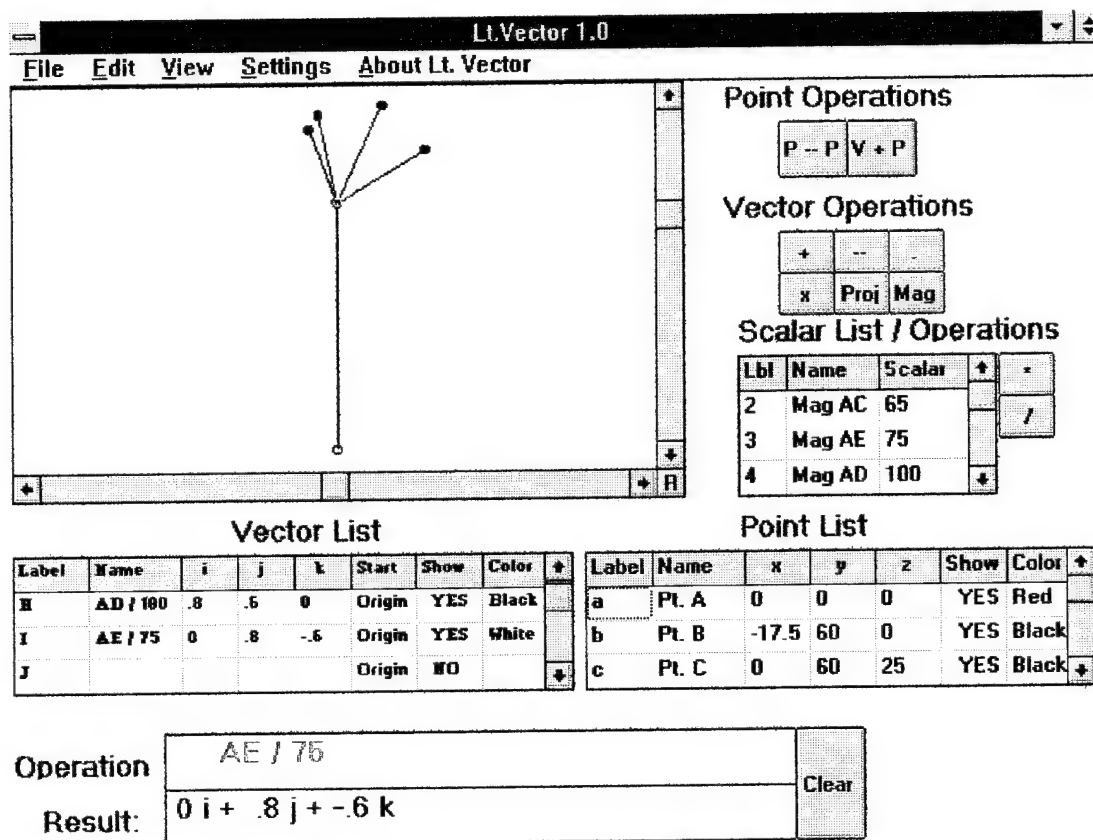nal forces in the process. If the problem is solved by hand, Lt. Vector can be used to check the solution, as shown in Figure 5.8, by multiplying the unit vectors by the force magnitudes.

Figure 5.8: Graphical solution to example 5.3.

5.5 Example 4: Cross-Product

Calculating the cross-product of two vectors is frequently required when doing any problem involving vector analysis. Mathematically it is a simple operation, and one that can be easily integrated into computer programs. However, as this example will reveal, there are some issues associated with the operation which are not easily resolved when designing a computer program.

In this example, the cross-product of two vectors is calculated and drawn. When a cross-product is calculated by hand, little thought is given to the location of the base point for the resultant vector; however, this is a concern when the operation is

rendered graphically. When the two vectors share a common base, it is logical to begin the vector resultant of the cross-product at the same point, as shown in Figure 5.9.
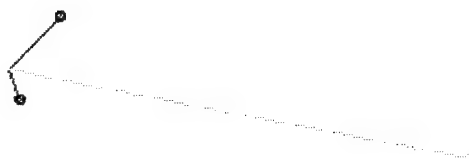
Figure 5.9: Cross-product of two vectors in Lt. Vector, common base point.

When the two vectors do not originate from the same location, the decision of where to place the resultant is not as clear, as shown in Figure 5.10

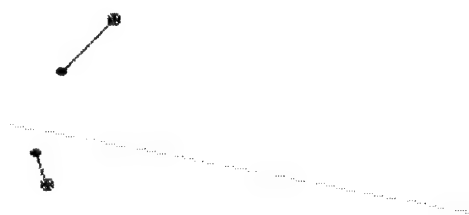

Figure 5.10: Cross-product of two vectors in Lt. Vector, different base points.

In the preceding figures, it is clear that the resultant vector is considerably larger than either of the other two vectors; fitting all the objects into the graphical output viewer required that the view be translated. This difficulty arises because the

resultant is a product, and consequently has different units than the original two vectors, but is plotted to the same scale.

5.6 Critique

The preceding example demonstrates the difficulty encountered in designing an effective vector visualization tool. Determining a meaningful location for the resultant vector depends upon the user's motivation for conducting the calculation. Lt. Vector allows cross-product calculation, but does not permit the user to state the reasons for doing so. Therefore, the base point for the resultant vector must be set in an arbitrary manner. Although it would be preferable to have the vector begin in the location that would make the most sense to the student, this is not a significant problem as the user can change a vector base point at any time.

The same example showed that it is possible for the resultant of an operation to be so much larger than the other objects on display that the perspective must be changed to view the resultant. The required change in perspective can be so great that it will render the other objects too small to be seen. This occurs because Lt. Vector does not track units; therefore, distance, force and moment vectors are all plotted on the same geometric scale. This problem with scaling reveals that unit management may be useful in this type of software, particularly if used to solve engineering problems. The difficulty in selecting appropriate base points for cross-products reveals that there often is no clear best answer to certain design issues.

The engineering mechanics problem in the third example revealed shortcomings in the modeling of scalars. As noted, it is possible to arrive at the solution by changing the magnitudes of the support vectors until the force loop closes upon itself. In Lt. Vector, this can only be accomplished through repeated scalar multiplication with changing scalar values, a rather tedious method. A program specifically designed to solve engineering mechanics problems should allow this type of calculation to be done easily. While the primary purpose of this program was to provide an interactive

environment for vector visualization, permitting this type of scalar manipulation could have increased effectiveness.

Chapter 6

SUMMARY AND CONCLUSIONS

The goal of this project was to construct a prototype software environment in which vectors, scalars and points could be entered, manipulated and observed in three-dimensional form to help entry-level engineering students develop 3-D visualization skills. Basic functionality was achieved in the resulting prototype, Lt. Vector, although critical areas that should be improved in later versions were identified.

The approach taken for this thesis was different than that taken in commonly used mathematical software, such as Matlab. Programs of this variety offer sophisticated numerical analysis and high-level graphics, but not in a readily interactive manner. Creating a program capable of numerically conducting analytical vector operations would not have been difficult, nor would designing one able to render 3-D vectors graphically. The challenge in achieving the project objective was that an environment had to be created granting users access to graphical, numerical and symbolic representations simultaneously, in an interactive manner.

In the course of this work, a number of critical issues regarding design of a successful vector visualization program were identified, and important lessons were learned in attempting to respond to these issues. The following paragraphs present a discussion of the issues and lessons learned, so as to provide a roadmap for future development.

Prior to additional programming, a thorough examination should be of the specific types of problems the resulting software would be used to solve, and the role

the program would play in the solution. Working through this problems by hand could lead the designer to new insights on internal software structure. The result of the work presented in this thesis could be described as a vector calculator; future work might examine the possibility of extending the scope to include a solver capable of managing and solving for unknown values.

If the program was to be used in a statics course, the underlying structure could be aligned more towards solving engineering mechanics problems, such as the third example of the preceding chapter. While it was determined early in this work that the resulting software should not be a "black-box" statics program giving students homework solutions, it was intended to be useful in solving these problems. In its current form, dynamic imaging allows students a greater visual understanding of a problem, but the student must already possess an understanding of the solution process. While this basic methodology is sound, it could be improved upon in later versions. For example, the prototype could be more useful in setting up the equations of equilibrium in a statics problem by allowing a user to sum the components of several selected vectors in a specified direction.

Developing visualization skills while teaching vector analysis can be facilitated by the employment of some type of computer-generated graphics. In this thesis, it was determined that providing dynamic viewing capabilities to the user would greatly aid in understanding rendered images. By permitting dynamic viewing, relatively low-level rendering and animation techniques could be employed effectively. While future designers could improve upon this technique through the use of more sophisticated graphics, this method was considered more successful than creating highly refined static images.

In creating a program requiring minimal introduction and instruction before use, design of the overall screen appearance is critical. The approach taken in the prototype was to situate many of the available options and operations such that they were immediately available to the viewer. While this enabled the user to conduct the

majority of operations without even having to use menus, it also resulted in a somewhat cluttered screen. Greater consideration should be given to appearance during the design phase to result in a less cluttered screen. One means of achieving this could involve placing more options off-screen on pull-down menus, rather than on permanent buttons.

It was decided early in this work to conduct all vector operations with the mouse. This created a more graphically-based program, with vector operations conducted by selecting points, scalars, vectors and the operation directly from the screen, without explicitly entering commands from the keyboard. Therefore, while the user had access to graphical and numerical representations, the only symbolic information was the feedback in the result window following each operation. This approach was limiting in that sequential vector operations had to be conducted in individual steps. For example, a user desiring to add vectors **A** and **B**, cross multiply with **C**, and the dot with vector **D,** must conduct three completely separate operations. Some thought should be directed towards finding a more succinct method of accommodating these type of operations. Construction of a parser allowing prompt-line entries should be investigated by future developers.

Data management, on both the interface and internal levels, is of critical importance to the successful design of this program. The spreadsheet-like tables of Lt. Vector were created to give the user ready access to vector, scalar and point information. Subsequent testing revealed these tables cumbersome to work with, however. The prompt-line approach taken in programs such as Matlab, where none of the information is immediately visible but can be retrieved with a typed command, does not seem appropriate. In an interactive environment, the user should have more immediate access to data than the prompt-line offers. A different approach that could be more successful would be to list only some information about the objects, with more available at a mouse click.

Internal data management in Lt. Vector also requires improvement, as demonstrated by the examples of the preceding chapter. Maintaining vector data in a format allowing users to easily change scalar magnitudes would facilitate solving engineering mechanics problems with this program. Additional improvements could result from tracking and storing vector units and types. Instead of rendering cross-products substantially larger than other vectors, the program could account for the fact that moments have different units, and draw them to a different scale. Moments could be visually distinguished easily by following the convention commonly used in textbooks of rendering with double arrowheads. Type tracking also would allow the program to prevent the user from meaningless calculations, such as crossing two moments.

A necessary step in the software design and construction process is that of rigorous testing. The scope of this work prevented conducting any large-scale, monitored testing. The logical step at this point would be to present a group of entry-level engineering students with typical homework problems, give them a minimal introduction to Lt. Vector, and determine how useful they found the prototype in working these problems. On a basic level, this type of testing and feedback would be useful in determining improvements to overall functionality. However, it would be extremely difficult to test and evaluate how well the prototype met the larger goal, aiding student development of visualization skills.

Ultimately, the goal of creating vector analysis software capable of developing three-dimensional visualization skills in entry-level engineering students appears achievable, though more work is required.

BIBLIOGRAPHY

Beer, Ferdinand P., and E. Russell Johnston, Jr. *Statics & Mechanics of Materials.*
New York: McGraw-Hill, Inc., 1992.

Brackeen, Debra, et al. *Apple Human Interface Checklist.* Cuperino: Apple
Computer, Inc., 1989.

Eberts, Ray E. *User Interface Design.* Englewood Cliffs: Prentice Hall, 1994.

Fournier, Alain and John Buchanan. "An Overview of Computer Graphics for
Visualization." *Computer Visualization: Graphics Techniques for Scientific
and Engineering Analysis.* ed. Richard S. Gallagher. Boca Raton: CRC Press,
1995. 17-51.

Gallagher, Richard S. "Future Trends in Scientific Visualization." *Computer
Visualization: Graphics Techniques for Scientific and Engineering Analysis.*
ed. Richard S. Gallagher. Boca Raton: CRC Press, 1995. 291-303.

Haber, R.B. "Visualization Techniques for Engineering Mechanics." *Computing
Systems in Engineering.* Vol. 1, No. 1. pp. 37-50, 1990.

McCuistion, Patrick J. "Static vs. Dynamic Visuals in Computer-Assisted Instruction."
Engineering Design Graphics Journal. Vol. 55, No. 2, pp. 1-33, 1991.

Mufti, Aftab A. *Elementary Computer Graphics.* Reston: Reston Publishing
Company, Inc., 1983.

Salmon, Rod and Mel Slater. *Computer Graphics: Systems & Concepts.* Wokingham:
Addison-Wesley Publishing Company, 1987.

Stasko, John T. "Animation in User Interfaces: Principles and Techniques." *User
Interface Software.* ed. Len Bass and Prasun Dewan. New York: John Wiley
& Sons, 1993. 81-101.

Thomas, George B. Jr. and Ross L. Finney. *Calculus and Analytic Geometry*. Reading: Addison-Wesley Publishing Company, 1989.

Watt, Alan. *Fundamentals of Three-Dimensional Computer Graphics*. Wokingham: Addison-Wesley Publishing Company, 1989.

Wiebe, Eric. "Scientific Visualization: An Experimental Introductory Graphics Course for Science and Engineering Students." *Engineering Design Graphics Journal*. Vol. 56, No. 1, pp. 39-44, 1992.

Appendix A

CODE SUMMARY

In creating a Visual Basic application, code associated with all screen objects is automatically generated. Additionally, there are the specific subroutines created by the software developer. In this summary, only the latter functions are presented, as the majority of code was contained in them. This appendix contains the header and a brief description of each of the general subroutines.

MAIN10.FRM: Majority of code is held here. Controls main interface form.

Sub Calculate (Operation As String, TypeOfOp As Integer): This function controls all user-specified calculations for the program. When any calculation button is pressed by the user, this function is called along with a tag identifying the desired operation. TypeOfOp is a variable which identifies what kind of operation it is (i.e., deals with points, vectors, etc.). The function conducts the desired operation through the use of the separate functions which exist for each operation. The items to be operated upon must have already been selected for this to work.

Sub ClearSelections (): Function is called to erase all selections.

Sub CordTrans (): This function calculates the coordinate transforms for the program and gives the global coordinates for the origin. It updates the three global arrays of viewer_x, _y and _z, which give the screen coordinates, and the focal length. It is also called when the form is loaded so that the arrays are initialized.

Sub DeletePoint (RowToDelete As Integer): Function allows user to delete a selected point. As input, it requires the row number of the point to delete. All it does is copy the contents of each grid square to the one above it.

Sub DeleteScalar (RowToDelete As Integer): Function allows user to delete a selected scalar. As input, it requires the row number of the scalar to delete. All it does is copy the contents of each grid square to the one above it.

Sub DeleteVector (RowToDelete As Integer): Function allows user to delete a selected vector. As input, it requires the row number of the vector to delete. All it does is copy the contents of each grid square to the one above it.

Function Dot (a() As Double, B() As Double) As Double: Function takes the dot product of two vectors. Input is two arrays with 3 components, output is the dot product.

Sub Draw (): Function controls all drawing subroutines and everything that appears in the graphical display window.

Sub DrawAxis (): This Function draws the axis system at the center of the screen. Will require the use of some global variables, namely the screen coordinate arrays.

Sub DrawCircle (X_Coord As Double, Y_Coord As Double, AColor As Double): This function will draw a circle, generally to be used in place of arrowheads to indicate vector direction. Requires 3 inputs, the screen coordinates and the color to be used in RGB format.

Sub DrawPoints (LastRow As Integer): This function draws the points from the point grid. All manipulations required to transform from 3D to 2D have already been completed in "TransformPoint." The required information is in "ArrayPoints."

Sub DrawVectors (LastRow As Integer): This function draws the vectors on the screen. It takes the required information, which was calculated by "TransformVector," from the array "ArrayVectEndPts."

Sub FindLastPoint (): Function determines the row number of the last point in the point grid and updates variable "LastPoint." This is done by cycling through the point list. When a point is found with no entry in any coordinate column, the loop stops with that row number entered in "LastPoint." LastPoint is initialized in Form_Load.

Sub FindLastScalar (): Function determines the row number of the last scalar in the scalar grid and updates variable "LastScalar." This is done by cycling through the scalar list. When a row is found with no entry the loop stops with that row number entered in "LastScalar." LastScalar is initialized in Form_Load.

Sub FindLastVector (): Function determines the row number of the last vector in the vector grid and updates variable "LastVector." This is done by cycling through the vector list. When a vector is found with no entry in any of the "I, j" or "k" columns, the loop stops with that row number entered in "LastVector." LastVector is initialized in Form_Load. This function is called when any change is made in the grid. Is called when any change occurs to the vector text box, and when the result of a vector calculation is sent to the grid. Note that if the user leaves a vector not completely filled, the program will regard that as an empty vector. If a calculation is made, the result will consequently overwrite a partially filled vector.

Sub GetStartPoint (): This function is called when the user selects the fifth column of GrVector. It allows the user to set the base point of a vector. Note that they start at the origin by default.

Sub InsertVector (RowWhereInsert As Integer): Function allows user to place a vector anywhere in middle of the list. Mainly of use when user wants to start an already existing vector at a vector they haven't entered, or something of that nature. Works by starting at bottom of list, and copying contents of each grid square down one spot

Function Mag (Vector() As Double) As Double: This function takes a vector as input, and returns its scalar magnitude.

Sub SelectVorP (X As Single, Y As Single, Shift As Integer): This function is called when the user clicks the right mousebutton on the display. It allows the user to select a vector or point by clicking on or near a displayed one. Basic algorithm was to first determine all the vectors and points that are near the user's click point. These points and vectors are then compared to determine which one is the closest.

Sub SetAngleRange (): This function sets the angle range depending on the global variable AngleRange, which can have 3 different values:
> 1: low of 0 and high of 1.57
> 2: low of -1.57 high of 0
> 3: low of 0 high of 3.14

Sub SetUpGrids (): This function is called only once, by function Form_Load to set up the three grids.

Function TransformPoint (ColNum As Integer, APoint() As Double, AColor As Integer) As Integer: This function calculates the 2D screen coordinates of a 3-D point given the 3-D coordinates. It writes the points to the global array, "ArrayPoints," along with other information about the point. It returns a value of true or false. If it returns true, it means that everything is okay. If it returns false, it means that a point was so large that it was essentially 'behind' the user, therefore the user's distance from the axis has to be increased by increasing viewZorigin and restarting Draw so that all the points are drawn to the same scale.

Function TransformVector (ColNum As Integer, APoint() As Double, AVector() As Double) As Integer: This function calculates the 2D screen coordinates of a vector given the 3-D coordinates of the starting point and the vector. It writes these points to the global array, "ArrayVectEndPts," along with other information about the vector. The function also returns a value of true or false. If it returns true, it doesn't really mean anything, only that all is well. However, if it returns false (to Draw, the calling function), it means that the value of viewZorigin is too small. The value is increased appropriately, then the important parts of Draw are repeated so that an accurate drawing is created.

Function Truncate (Num As Double) As Double: This is a simple function which truncates a number down to significant figures past the decimal. Rounds last digit appropriately. Algorithm is to multiply the number by 10 to the desired number of digits (the accuracy), round off anything past the decimal, then divide through by the accuracy once more. A few more lines of code are required to round off correctly, and to deal with the numerical sign. Accuracy is controlled by global variable Precision, which user can change at run time as desired.

Sub VectAdd (A() As Double, B() As Double): Function will add two vectors together. It will take two input arrays but does not directly return an answer. Since VB won't return arrays, it will write its answer to a reserved global temporary array, TempVectAdd.

Sub VectCross (A() As Double, B() As Double): Function computes the cross product of two vectors. Writes answer to TempVectCross.

Sub VectProj (A() As Double, B() As Double): Function takes the projection of one vector in the direction of another. Works as Projection of vector A onto vector B.

Sub VectScalarMult (A() As Double, Scalar As Double): Function will multiply a vector by a scalar, returning result to array TempVectScalarMult.

Sub VectSub (A() As Double, B() As Double): Function subtracts one Vector from another. Writes results to temporary array TempVectSub

Sub WriteAnswerPoint (Answer() As Double): Function writes the Point result of a vector operation to the first empty slot in the list of Points (GrPoint). Additionally, the calculation and result are written to the result window.

Sub WriteAnswerScalar (Answer As Double): Function takes care of Scalar Results. Writes answer to answer window and to the list of scalars, GrScalar.

Sub WriteAnswerVector (Answer() As Double, RowNum): Function writes the vector result of a vector operation to the first empty slot in the list of vectors (GrVector). Additionally, the calculation and result are written to the result window.

Sub WriteCmdVectAddSub (Operation As String, ArrayOfRows() As Integer): Function is called to write out the names of the vectors which were added or subtracted. This separate function is required for these two operations because they are the only two which can be conducted with more than two vectors.

Sub WriteCommandScalar (Operation As String, RowA, RowB): If an operation results in a scalar answer, this function writes the operation to the command window.

Sub WriteCommandVector (Operation As String, RowA, RowB): Function will write the command that just occurred to the command window so that the user will know what was just accomplished.

MOMAIN.BAS: Contains code accessible by all parts of the function, i.e. global subroutines.

Function CheckNumericalEntry (TheEntry As String) As Integer: Function is used in FoStartPoint and FoMain to check the input of entry boxes. The entry boxes already limit which characters can be entered, but make no logic checks. For example, a user can enter '-1.34.4-1', makes no sense, but also uses no illegal characters. This will prevent this sort of error from occurring.

Sub ErrorEntry (TheEntry): Function displays an error message

Function FindMatchPoint (PointName) As Integer: Function is used to determine the row number in GrPoint of a Point, given its label. It returns the row number of the Point. If no match is found, it returns a negative number.

Function FindMatchVector (VectorName) As Integer: Function is used to determine the row number in GrVector of a vector, given its label. It returns the row number of the vector. If no match is found, it returns a negative number.

Sub IsItOk (): This function will be used in FoStartPoint, the form that allows the user to select the starting point for a vector. Primarily, it keeps the "OK" button disabled until the user has entered an acceptable start location.

FOSTARTPOINT.FRM: Form is used to obtain vector base points.

Function CheckCharOk (Char As Integer): Function is used to check if user-input characters are "acceptable," in that they are numbers, decimal point, enter, backspace, or negative signs.

A Visualization Tool for Engineering Vector Analysis

Byron L. Miranda

A prototype computer tool was designed to support an interactive, visual approach to the learning of vector analysis. The objective was to construct a prototype environment in which vectors, scalars and points could be entered, manipulated and observed in three-dimensional form in order to investigate new mechanisms for linking mathematical abstractions to intuitive, geometric thinking at an early stage in a student's education. Vector analysis is typically an integral part of an engineering student's introduction to 3-D thinking. When first introduced, vectors and the results of vector operations often are presented geometrically so that the student may gain some sense of what they actually "look" like. This is often hindered, however, by the two-dimensional limitations of paper and chalkboard, and by the time required to complete the mathematical calculations and graphically render them in a meaningful way. Computers permit rapid vector calculations and allow quasi-real-time graphical presentation of the results of these calculations. Therefore, students could be able to see the results of their work as it is done in a richer and dynamic fashion. Basic functionality was achieved in the resulting prototype, although critical areas that should be improved in later versions were identified Ultimately, the goal of creating vector analysis software capable of developing three-dimensional visualization skills in entry-level engineering students appears achievable, though more work is required.

BIBLIOGRAPHY

Beer, Ferdinand P., and E. Russell Johnston, Jr. *Statics & Mechanics of Materials.* New York: McGraw-Hill, Inc., 1992.

Brackeen, Debra, et al. *Apple Human Interface Checklist.* Cuperino: Apple Computer, Inc., 1989.

Eberts, Ray E. *User Interface Design.* Englewood Cliffs: Prentice Hall, 1994.

Fournier, Alain and John Buchanan. "An Overview of Computer Graphics for Visualization." *Computer Visualization: Graphics Techniques for Scientific and Engineering Analysis.* ed. Richard S. Gallagher. Boca Raton: CRC Press, 1995. 17-51.

Gallagher, Richard S. "Future Trends in Scientific Visualization." *Computer Visualization: Graphics Techniques for Scientific and Engineering Analysis.* ed. Richard S. Gallagher. Boca Raton: CRC Press, 1995. 291-303.

Haber, R.B. "Visualization Techniques for Engineering Mechanics." *Computing Systems in Engineering.* Vol. 1, No. 1. pp. 37-50, 1990.

McCuistion, Patrick J. "Static vs. Dynamic Visuals in Computer-Assisted Instruction." *Engineering Design Graphics Journal.* Vol. 55, No. 2, pp. 1-33, 1991.

Mufti, Aftab A. *Elementary Computer Graphics.* Reston: Reston Publishing Company, Inc., 1983.

Salmon, Rod and Mel Slater. *Computer Graphics: Systems & Concepts.* Wokingham: Addison-Wesley Publishing Company, 1987.

Stasko, John T. "Animation in User Interfaces: Principles and Techniques." *User Interface Software.* ed. Len Bass and Prasun Dewan. New York: John Wiley & Sons, 1993. 81-101.

Thomas, George B. Jr. and Ross L. Finney. *Calculus and Analytic Geometry.* Reading: Addison-Wesley Publishing Company, 1989.

Watt, Alan. *Fundamentals of Three-Dimensional Computer Graphics.* Wokingham: Addison-Wesley Publishing Company, 1989.

Wiebe, Eric. "Scientific Visualization: An Experimental Introductory Graphics Course for Science and Engineering Students." *Engineering Design Graphics Journal.* Vol. 56, No. 1, pp. 39-44, 1992.

A Visualization Tool for Engineering Vector Analysis

Byron L. Miranda

A prototype computer tool was designed to support an interactive, visual approach to the learning of vector analysis. The objective was to construct a prototype environment in which vectors, scalars and points could be entered, manipulated and observed in three-dimensional form in order to investigate new mechanisms for linking mathematical abstractions to intuitive, geometric thinking at an early stage in a student's education. Vector analysis is typically an integral part of an engineering student's introduction to 3-D thinking. When first introduced, vectors and the results of vector operations often are presented geometrically so that the student may gain some sense of what they actually "look" like. This is often hindered, however, by the two-dimensional limitations of paper and chalkboard, and by the time required to complete the mathematical calculations and graphically render them in a meaningful way. Computers permit rapid vector calculations and allow quasi-real-time graphical presentation of the results of these calculations. Therefore, students could be able to see the results of their work as it is done in a richer and dynamic fashion. Basic functionality was achieved in the resulting prototype, although critical areas that should be improved in later versions were identified Ultimately, the goal of creating vector analysis software capable of developing three-dimensional visualization skills in entry-level engineering students appears achievable, though more work is required.

BIBLIOGRAPHY

Beer, Ferdinand P., and E. Russell Johnston, Jr. *Statics & Mechanics of Materials.* New York: McGraw-Hill, Inc., 1992.

Brackeen, Debra, et al. *Apple Human Interface Checklist.* Cuperino: Apple Computer, Inc., 1989.

Eberts, Ray E. *User Interface Design.* Englewood Cliffs: Prentice Hall, 1994.

Fournier, Alain and John Buchanan. "An Overview of Computer Graphics for Visualization." *Computer Visualization: Graphics Techniques for Scientific and Engineering Analysis.* ed. Richard S. Gallagher. Boca Raton: CRC Press, 1995. 17-51.

Gallagher, Richard S. "Future Trends in Scientific Visualization." *Computer Visualization: Graphics Techniques for Scientific and Engineering Analysis*. ed. Richard S. Gallagher. Boca Raton: CRC Press, 1995. 291-303.

Haber, R.B. "Visualization Techniques for Engineering Mechanics." *Computing Systems in Engineering*. Vol. 1, No. 1. pp. 37-50, 1990.

McCuistion, Patrick J. "Static vs. Dynamic Visuals in Computer-Assisted Instruction." *Engineering Design Graphics Journal*. Vol. 55, No. 2, pp. 1-33, 1991.

Mufti, Aftab A. *Elementary Computer Graphics*. Reston: Reston Publishing Company, Inc., 1983.

Salmon, Rod and Mel Slater. *Computer Graphics: Systems & Concepts*. Wokingham: Addison-Wesley Publishing Company, 1987.

Stasko, John T. "Animation in User Interfaces: Principles and Techniques." *User Interface Software*. ed. Len Bass and Prasun Dewan. New York: John Wiley & Sons, 1993. 81-101.

Thomas, George B. Jr. and Ross L. Finney. *Calculus and Analytic Geometry*. Reading: Addison-Wesley Publishing Company, 1989.

Watt, Alan. *Fundamentals of Three-Dimensional Computer Graphics*. Wokingham: Addison-Wesley Publishing Company, 1989.

Wiebe, Eric. "Scientific Visualization: An Experimental Introductory Graphics Course for Science and Engineering Students." *Engineering Design Graphics Journal*. Vol. 56, No. 1, pp. 39-44, 1992.